WEB VULNERABILITIES SCANNER

by
Fatih Furkan Uslu

Engineering Project Report

Yeditepe University
Faculty of Engineering
Department of Computer Engineering
2024

WEB VULNERABILITIES SCANNER

APPROVED BY:

Assist. Prof. Dr. Dionysis Goularas . . . . . .
(Supervisor)

Assist. Prof. Dr. Esin Onbaşıoğlu . . . . . .

Prof. Dr. Sezer Gören Uğurdağ . . . . . .

DATE OF APPROVAL: .../.../2021

ACKNOWLEDGEMENTS

First of all I would like to thank my advisor ... for his guidance and support throughout my project.

Also I would like to thank my parents for their support and encouragement throughout my education up to the present.

ABSTRACT

WEB VULNERABILITIES SCANNER

Automatic music transcription is an ongoing research field as a branch of music information retrieval and digital signal processing. It can be used for transcribing high amounts of music at once or categorizing and indexing music on databases for pattern matching or machine learning. It can be a challenge to create a general purpose algorithm as pitch is a concept related to human perception and harder for complex signals of polyphonic music. This project focuses on monophonic music only, by explaining several existing approaches and demonstrating the implementation of one of them.

ÖZET

TITLE OF THE PROJECT

Otomatik müzik transkripsiyonu onyıllardır sürmekte olan bir araştırma alanıdır. Müzikten bilgi edinme ve dijital sinyal işlemenin bir alt dalıdır. Bir seferde çok sayıda transkripsiyon yapma, veri tabanları üzerinde kategorizasyon ve indeksleme ile makine öğrenmesi için kullanılabilir. Duyulan bir nota insan kulağıyla ilgili bir durum olduğu için belirlenmesini sağlayacak kesin bir algoritma ortaya konulması sorunludur. Karmaşık polifonik sesler için daha sorunludur. Bu raporda, var olan birkaç yöntemden söz edilmekte ve içlerinden birinin uygulanışı gösterilmektedir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS/ABBREVIATIONS

δ_t	Threshold
f_0	Fundamental frequency
$\circ$	Hadamard Product
μ	Magnitude
ReLU	Rectifier Function
ρ	Resampling Function
S	Spectrogram
$\hat{S}$	HPS Applied Spectrogram
φ	Phase
X	Complex STFT Output

ABI	Application Binary Interface
CLI	Command Line Interface
CPU	Central Processing Unit
DFT	Discrete Fourier transform
DSP	Digital Signal Processing
FFT	Fast Fourier Transform
FFI	Foreign Function Interface
GPU	Graphics Processing Unit
MIDI	Musical Instrument Digital Interface
MIR	Music Information Retrieval
PDA	Pitch Detection Algorithm
STFT	Short Time Fourier Transform
VRAM	Video Ram (Random Access Memory)
WAV	Waveform Audio File Format

1. INTRODUCTION

1.1. NEED of Security

Why do we need security ? For privacy reasons or to feel safe ? Maybe for economic reasons… It's actually all of them. Security is a fundamental aspect of human existence, dating back to ancient civilizations where tribes protected their resources from external threats. They protected their food from animals, other tribes, and weather conditions. They protected their women, children, and their elders from enemy tribes. They protected their caves from extreme weather conditions and other creatures. They all wanted a secure life. They wanted to live without danger. To protect their cave and food, they prepared traps. Also, while attacking their enemies, they were careful about the traps they knew while falling for the ones they did not know. And it wasn't just old tribes but also every human from every era have been living this. From caves to World War 1 to present, security has always been humanity's instinctive desire.

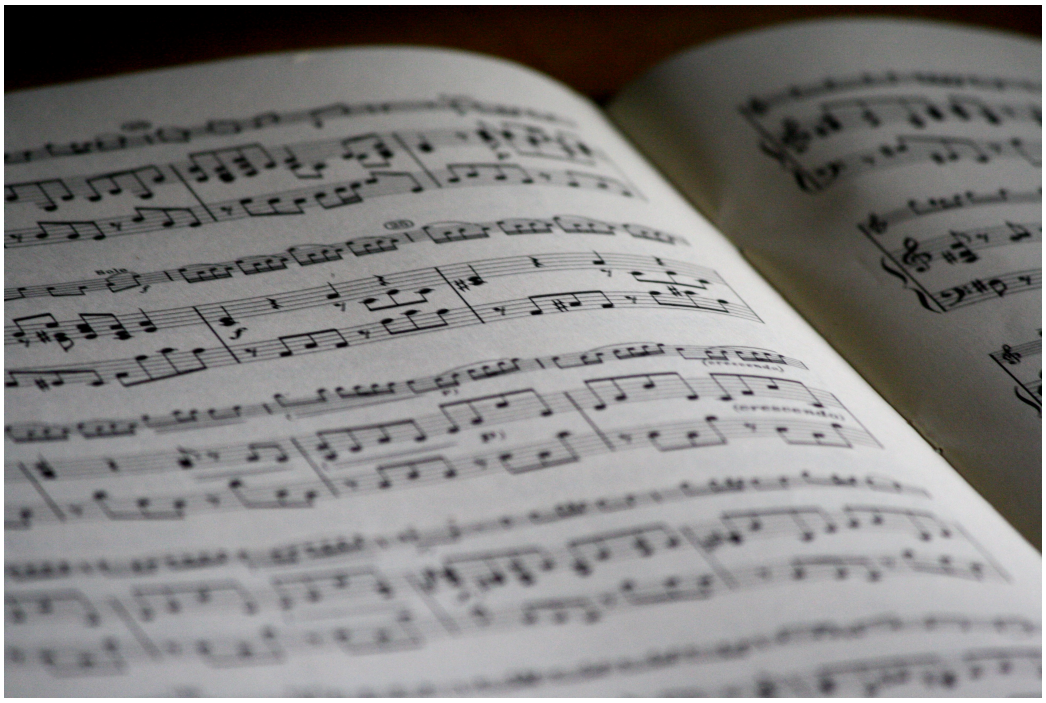

Figure 1.1. Sheet Music

This was more like protection of life and property. What about privacy ? No need to think about only our secrets. It's about having a private life, a personal distance. Every individual

has the right to have privacy. It's completely normal and it should be.

1.1.1. The Evolution of Security: From Physical Protection to Digital Safeguards

In today's interconnected world, the notion of security has expanded beyond physical protection to include digital security. With the advent of the internet, our lives have become increasingly digitized, making cybersecurity a crucial aspect of modern living. Cybersecurity is essential for protecting sensitive information, ensuring privacy, and maintaining the integrity of systems that support our daily activities and economic transactions.

Web applications, which are ubiquitous in modern society, are a prime target for cyber threats. From online banking and e-commerce platforms to social media and healthcare portals, web applications handle vast amounts of sensitive data. This makes them attractive targets for cybercriminals looking to exploit vulnerabilities for financial gain, identity theft, or malicious disruption.

The complexity and constant evolution of web applications introduce numerous potential vulnerabilities. These vulnerabilities can be exploited through various methods, such as SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF). Each of these attack vectors can compromise the security and privacy of users, leading to severe consequences for individuals and organizations alike.

To address these challenges, web vulnerability scanners have become indispensable tools in the arsenal of cybersecurity professionals. These scanners automate the process of detecting and analyzing security weaknesses in web applications. By simulating attacks and identifying potential vulnerabilities, they help developers and security teams to fortify their applications against real-world threats.

The purpose of this report is to provide an in-depth analysis of our web vulnerabilities scanner app. This app is designed to proactively identify and mitigate security risks in web applications. It employs advanced scanning techniques to detect a wide range of vulnerabilities, providing detailed reports and actionable insights for remediation.

In the following sections, we will explore the features and capabilities of our web vulnerabilities scanner. We will discuss its architecture, scanning methodologies, and the types of vulnerabilities it can detect. Additionally, we will highlight case studies and real-world applications to demonstrate its effectiveness in enhancing web security.

By understanding the importance of web security and the role of vulnerability scanners, we can appreciate the necessity of integrating such tools into the development and maintenance of web applications. This report aims to shed light on the critical function of our web vulnerabilities scanner in safeguarding digital assets and ensuring a secure online environment for users.

1.2. Terms

- *Cybersecurity* The practice of protecting systems, networks, and programs from digital attacks. These attacks are usually aimed at accessing, changing, or destroying sensitive information; extorting money from users; or interrupting normal business processes.
- *Web Application* A software application that runs on a web server and can be accessed via a web browser. Examples include online banking platforms, e-commerce sites, social media networks, and healthcare portals.
- A *Vulnerability* , A weakness or flaw in a system that can be exploited by a threat actor, such as a hacker, to gain unauthorized access to or perform unauthorized actions on a system.
- *SQL Injection* A code injection technique that exploits a security vulnerability in a web application's software by inserting malicious SQL statements into an entry field for execution, thereby allowing the attacker to interfere with the queries that an application makes to its database.
- *Cross-Site Scripting (XSS)* A type of security vulnerability typically found in web applications. XSS allows attackers to inject malicious scripts into content from otherwise trusted websites. This can lead to unauthorized actions such as hijacking user sessions, defacing websites, or redirecting users to malicious sites.
- *Web Vulnerability Scanner* A software tool that automates the process of checking web applications for security vulnerabilities. It scans web applications to detect common vulnerabilities and provides reports that help developers and security teams to address these issues.
- *Threat Actor* An entity that is responsible for an event that has the potential to impact the security of an organization's systems and data. Threat actors can be individuals, groups, or organizations.

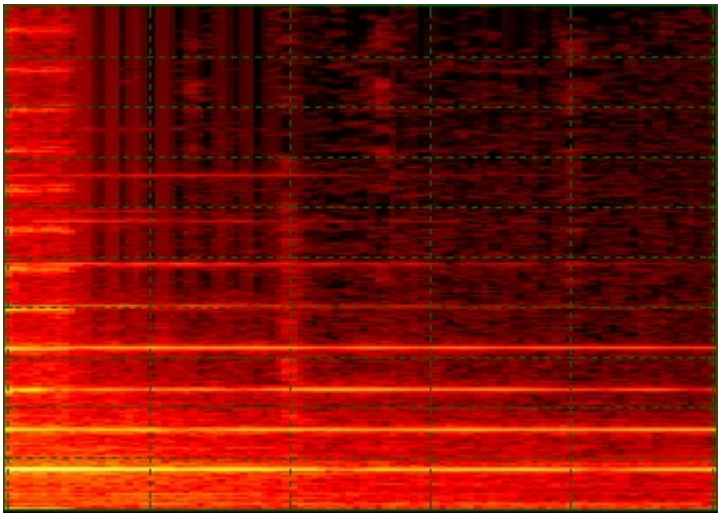

Figure 1.2. Harmonic Spectrum on the Spectrogram

- *Remediation*, the process of correcting a vulnerability or issue to prevent it from being exploited. In the context of web security, remediation involves making changes to code, configurations, or processes to fix identified vulnerabilities.

1.3. Motivation

The motivation behind developing a web vulnerabilities scanner stems from the increasing dependence on web applications in virtually every aspect of modern life. As digital transformation accelerates, ensuring the security of web applications becomes crucial to protect sensitive information, maintain user trust, and prevent economic losses. The complexity and sophistication of cyber threats necessitate advanced tools that can proactively identify and mitigate vulnerabilities in web applications.

1.3.1. Increasing Cyber Threats

The number of cyber threats targeting web applications has been rising steadily. Cybercriminals employ sophisticated methods to exploit vulnerabilities, leading to data breaches, financial losses, and reputational damage for organizations. High-profile cyber attacks demonstrate the potential scale and impact of these threats, highlighting the urgent need for effective security measures. Our web vulnerabilities scanner aims to address these challenges by providing a comprehensive tool to detect and manage security risks before they can be exploited.

1.3.2. As a Basis for Other Applications

Automatic Music Transcription is a form of Music Information Retrieval (MIR). MIR techniques are used in various fields. Retrieved information can be used for creating databases to categorize music with indexing and pattern matching or training machine learning algorithms and music generation. Also, it can be said that converting an audio file into MIDI is a form of compression.

1.4. Scope and Limitations

The scope of our web vulnerabilities scanner encompasses the detection and analysis of a wide range of security vulnerabilities in web applications. It is designed to identify common vulnerabilities such as SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF), among others. The scanner provides detailed reports and recommendations for remediation, assisting developers and security teams in addressing identified issues. However, there are limitations to the tool. It may not detect all possible vulnerabilities, particularly zero-day exploits that have not yet been documented or widely recognized. Additionally, while the scanner can identify vulnerabilities, the actual remediation requires human intervention and expertise. False positives and false negatives are also possible, necessitating a thorough review and validation by security professionals.

1.5. Problem Definition

Web applications are increasingly becoming targets for cyber attacks due to the valuable data they handle and their critical role in business operations. Despite the awareness of these risks, many organizations struggle to maintain robust security measures, often due to a lack of resources, expertise, or comprehensive tools. The main problem addressed by our web vulnerabilities scanner is the need for an efficient, automated solution that can identify and mitigate security vulnerabilities in web applications, thereby reducing the risk of cyber attacks and ensuring compliance with regulatory standards.

1.6. Requirements

To effectively address the problem of web application security, our web vulnerabilities scanner must meet the following requirements:

Comprehensive Vulnerability Detection: The scanner must be capable of identifying range of security vulnerabilities, including but not limited to SQL injection, XSS.

Detailed Reporting: The scanner should provide detailed reports that include the nature of detected vulnerabilities, their potential impact, and specific recommendations for fixing them.

Regular Updates: The tool must be regularly updated to include the latest known vulnerabilities and scanning techniques to stay effective against new and emerging threats.

2. BACKGROUND

The increasing prevalence of cyber attacks on web applications has driven significant research and development in the field of web security. Various tools and methodologies have been developed to identify and mitigate security vulnerabilities, with web vulnerability scanners being a critical component of these efforts. This section provides an overview of previous works in the domain of web vulnerability scanning, highlighting key advancements and persistent challenges.

2.1. Previous works

2.1.1. Early Web Vulnerability Scanners

The first generation of web vulnerability scanners emerged in the late 1990s and early 2000s, as the internet began to flourish and web applications became more widespread. Early tools, such as Nikto and Paros Proxy, were rudimentary in nature, focusing primarily on identifying basic security issues like outdated software versions and simple misconfigurations. These scanners laid the groundwork for more sophisticated tools but were limited in their ability to detect complex vulnerabilities.

2.1.2. Evolution of Scanning Techniques

As web applications grew more complex, so did the methods for attacking them. This led to the development of more advanced web vulnerability scanners that incorporated automated testing for a broader range of vulnerabilities. Tools such as Nessus and Acunetix introduced more sophisticated scanning techniques, including the ability to detect SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF) vulnerabilities. These scanners utilized techniques like fuzzing, which involves inputting unexpected data into web forms to uncover security flaws.

2.1.3. Integration with Development Lifecycles

In recent years, there has been a significant shift towards integrating security scanning into the software development lifecycle (SDLC). This approach, known as DevSecOps, emphasizes the importance of incorporating security practices from the earliest stages of development. Tools like OWASP ZAP (Zed Attack Proxy) and Burp Suite have become popu-

lar for their ability to integrate with continuous integration/continuous deployment (CI/CD) pipelines, allowing for automated security testing throughout the development process. This integration helps identify and address vulnerabilities early, reducing the risk of security issues in production environments.

2.1.4. Machine Learning and AI in Web Vulnerability Scanning

The latest advancements in web vulnerability scanning leverage machine learning and artificial intelligence to enhance the accuracy and efficiency of detecting security issues. Machine learning algorithms can analyze vast amounts of data to identify patterns and predict potential vulnerabilities. Tools like DeepExploit and Microsoft's Project Springfield utilize AI to automate the detection of vulnerabilities, minimizing false positives and identifying zero-day exploits more effectively.

2.1.5. Comparative Studies and Benchmarking

Numerous comparative studies and benchmarking efforts have been conducted to evaluate the effectiveness of different web vulnerability scanners. These studies often assess factors such as detection accuracy, scan speed, ease of use, and the ability to integrate with other tools. For instance, a study by the Open Web Application Security Project (OWASP) compared various scanners and highlighted the strengths and weaknesses of each, providing valuable insights for organizations choosing a tool.

2.1.6. Persistent Challenges

Despite significant advancements, web vulnerability scanners continue to face several challenges:

1.False Positives and Negatives: Ensuring the accuracy of vulnerability detection remains a critical issue. False positives can overwhelm security teams with unnecessary alerts, while false negatives can leave critical vulnerabilities undetected.

2.Zero-Day Vulnerabilities: The dynamic nature of web applications and the continuous emergence of new vulnerabilities mean that scanners must constantly evolve. Detecting zero-day vulnerabilities—unknown flaws that have not yet been addressed—remains particularly challenging.

3.Performance and Scalability: As web applications become more complex, scanning them comprehensively without impacting performance requires significant computational resources. Scalability is a crucial factor for large organizations with extensive web assets.

4.Integration and Usability: While integrating scanners into the SDLC is beneficial, achieving seamless integration and ensuring usability for developers and security teams can be difficult. Tools must be user-friendly and fit well into existing workflows.

2.1.7. Conclusion

The evolution of web vulnerability scanners reflects the growing importance of web application security in an increasingly digital world. From basic tools detecting simple misconfigurations to advanced AI-driven solutions capable of identifying sophisticated attacks, the field has made significant strides. However, ongoing challenges such as false positives, zero-day vulnerabilities, performance issues, and integration difficulties underscore the need for continued innovation and improvement.

Our web vulnerabilities scanner builds on the foundation of previous works, addressing these challenges with cutting-edge techniques and a user-centric design. By learning from past advancements and persistent issues, we aim to provide a robust, efficient, and accurate tool for enhancing web application security.

3. ANALYSIS

The complete process can be seen on Figure 3.7

3.1. Filtering

First, a high pass filter with a chosen cut off frequency is applied to the digital signal using discrete convolution (Equation 3.1) This process eliminates low frequencies which we might not be interested in and provides noise reduction.

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m] \tag{3.1}$$

3.2. Spectrograms

Then, from the resulting filtered samples two separate spectrograms are calculated, one with relatively large and another with relatively small window size later to be combined. The need for two separate spectrograms and the combination technique is explained in Section 3.3. In discrete-time STFT, input samples are divided into overlapping chunks with a chosen window size and a step size. Applying Fast Fourier Transform on each of these chunks and appending them to a matrix as columns will result in a two dimensional matrix of complex numbers as the STFT output expresses both phase and magnitude. Only the bottom half of the frequency axis is used because of its symmetry. Calculating a spectrogram is done with the following discrete-time short time Fourier transform:

$$X(m, w) = \sum_{n=-\infty}^{\infty} x[n]w[n-m]e^{-jwn} \tag{3.2}$$

where x is the input signal and w is a Gaussian curve window, then getting the magnitudes from the complex values with:

$$S(\tau, w) \equiv |X(\tau, w)| \tag{3.3}$$

where τ is the time frame and w is the frequency bin. Similarly, $arctan(X(\tau, w))$ gives the phases.

Require audio samples, window size, step size
$audio\ length \Leftarrow$ number of audio samples
$spectrogram\ length \Leftarrow (audio\ length - (window\ size - step\ size))/step\ size$
$spectrogram\ height \Leftarrow window\ size$
Form the $spectrogram\ height \times spectrogram\ length$ matrix **X**

$i \Leftarrow 0, j \Leftarrow 0$
while $i < audio\ length - window\ size$ **do**
  $chunk \Leftarrow$ audio samples from index i to $i + window\ size$
  $chunk \Leftarrow chunk \circ (hamming\ function$ of length $window\ size)$
  $new\ column \Leftarrow FFT(chunk)$
  $j^{th}\ column\ of X \Leftarrow new\ column$
  $i \Leftarrow i + step\ size, j \Leftarrow j + 1$
end while

Take rows $0\ to\ (window\ size)/2$ from X

Figure 3.1. Short Time Fourier Transform Algorithm.

3.3. Combined Spectrogram

Due to the nature of STFT a resulting spectrogram can either have good frequency resolution or good time resolution depending on the window size. This shortcoming can be mediated by calculating two spectrograms called wideband and narrowband, where wide and narrow refer to the window size used in the STFT and multiplying the corresponding pixel values of the two creating a combined spectrogram [1] which preserves the visual features associated with both. Characteristics of all the spectrograms can be seen in example figures in Section 4. In practice this can be achieved with the following:

After getting the two spectrograms A and B such that:

$$A_{m,n} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}, B_{p,q} = \begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,q} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,q} \\ \vdots & \vdots & \ddots & \vdots \\ b_{p,1} & b_{p,2} & \cdots & b_{p,q} \end{pmatrix}, \begin{cases} m > p \\ n < q \end{cases} \tag{3.4}$$

they are both resized to match the same shape of height m and length q by calculating the missing values using bilinear interpolation. Then the element-wise multiplication (Hadamard product) takes place like so:

$$C_{m,q} = A_{m,q} \circ B_{m,q} \tag{3.5}$$

where C is the $m \times q$ matrix i.e. the combined spectrogram.

3.4. Harmonic Product Spectrum

HPS is a process used to make the fundamental frequencies stand out by using the STFT output. This process involves two steps that is repeated as needed. First step is compression with resampling the spectrogram along the frequency axis and second step is multiplying the resulting compressed spectrogram with the original one, element by element by aligning the starting indices. In the first iteration magnitude spectrogram is compressed to half its size, on the next iteration to the third of its size and so on. Compression by an integer makes the partial frequencies of a pitch start to line up on the same frequency as seen on Figure 3.3. The frequency which the partials line up on and make its magnitude higher because of the multiplication is the fundamental (f_0). $X(w_n)$ in Figure 3.3 are showing the magnitudes of the frequencies in a single time frame.

$$\hat{S}(\tau, w) = \prod_{r=1}^{R} \rho_r(\tau, w), \;\; w = \lfloor w_0/R \rfloor \tag{3.6}$$

where R is the HPS rate, meaning that how many times the original (combined spectrogram from Section 3.3) is compressed and w_0 is the original spectrograms height. ρ_r is the resampled spectrogram with a new height of r^{th} of the original and calculated as:

$$\rho_r(\tau, w) = \frac{\sum_{i=0}^{r} S(\tau, rw + i)}{r} \tag{3.7}$$

Require rate, spectrogram
Form the $spectrogram.height/rate \times spectrogram.length$ matrix **H**
for $r = 2$ to R **do**
  $downsampled \Leftarrow$ downsample spectrogram to r^{th} of its size
  $d \Leftarrow$ take rows 0 to $spectrogram.height/rate$ from downsampled
  $H \Leftarrow H \circ d$
end for

Figure 3.2. Harmonic Product Spectrum Algorithm

HPS method is fast, inexpensive and reasonably resistant to noise but becuase of the human picth perception being logarithmic it falls short for low frequencies. It is also dependant on the FFT resolution [2] that was used for its accuracy.

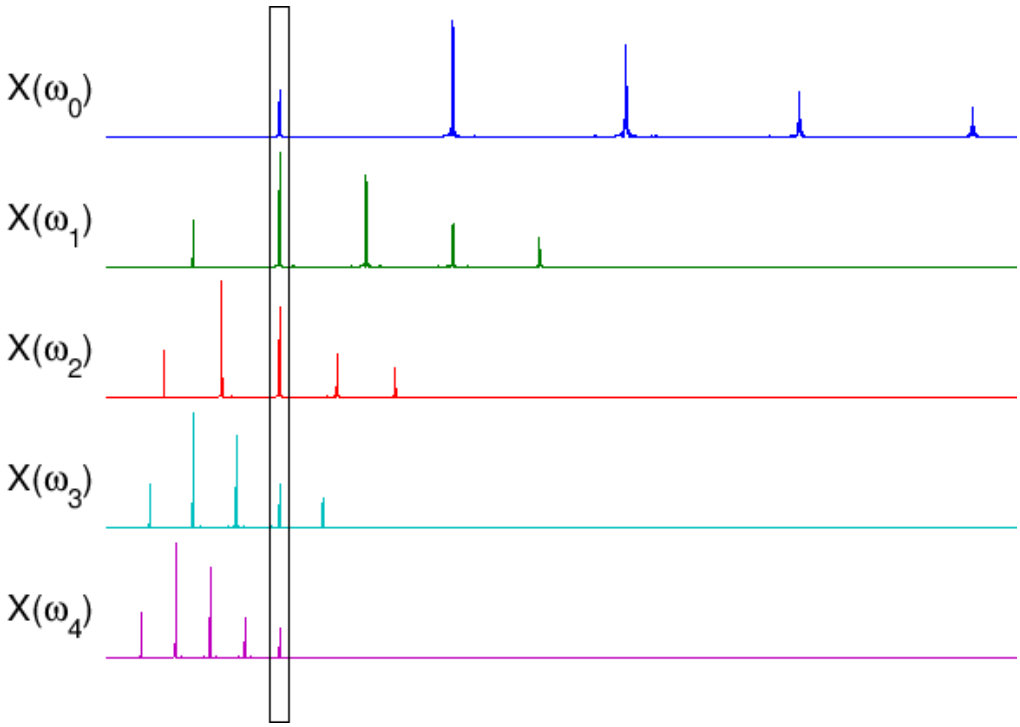

Figure 3.3. Harmonic Product Spectrum

3.5. Getting Frequencies

A fundamental frequency f_0 for each time frame is the frequency with the maximum amplitude (Equation 3.8) after HPS is applied .

$$f_0(t) = \max(f_t(x)) \tag{3.8}$$

Require spectrogram
Form an array F with length $spectrogram.length$
for $i = 0$ to $spectrogram.length$ **do**
  F[i] = index of maximum value from spectrogram[i]
end for

Figure 3.4. Getting Frequencies Algorithm

3.5.1. Frequency to Note Conversion

After a list of frequencies which every value corresponds to a time frame are acquired as seen in Section 3.5, this information is combined with the onset detection which is explained in section 3.6. For every note duration which we know the starting and finishing times from the onset detection, a slice from the frequency list corresponding to that time interval is used. This time interval is translated into a single frequency value by taking the mode of the slice and later translated again into a semitone difference using Equation 3.10

To get the frequency of a note [3] on an equal tempered scale following formula is used:

$$f_n = f_a * (\sqrt[12]{2})^n \tag{3.9}$$

where f_a is the known frequency of a fixed note of our choosing. A common choice is A_4 which is the A above middle C with a frequency of 440 Hz. n is the number of half steps (semitones) away from the fixed note. A higher note would make n positive and similarly a lower note would make it negative. f_n is the frequency of the note n half steps away from f_a

Since we have frequencies and need the notes, Equation 3.9 has to be reversed as such:

$$n = log_{(\sqrt[12]{2})}\left(\frac{f_n}{f_a}\right) \tag{3.10}$$

Knowing the n value, we can use it to get the notes' pitches and octaves from a table such as Table 3.1 or directly calculating.

Table 3.1. Notes' semitone distances to A.

-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2
C_4	$C_4\sharp/B_4\flat$	D_4	$D_4\sharp/E_4\flat$	E_4	F_4	$F_4\sharp/G_4\flat$	G_4	$G_4\sharp/A_4\flat$	A_4	$A_4\sharp/B_4\flat$	B_4

3.6. Onset Detection

Onset detection gives the information when a note begins and the last one finishes including rests. It works best with signals where an instrument with high attack and distinct note changes is used. There are energy and phase based approaches for onset detection [4]. For instruments with strong attacks, an increase in energy is expressive and sufficient for onset detection. As for phase, during the steady-state part of a signal most values will be concentrated around zero creating a sharp distribution but on onset points this distribution gets wider and less sharp. Energy approaches fail on the lack of pronounced energy increases and phase approaches are susceptible to phase distortion and to the variations introduced by the phase of noisy components. Combining energy and phase information [4] together a better result can be achieved.

Two narrowband spectrograms of magnitude and phase denoted as μ and φ respectively in Equation 3.11 are used for calculating an onset detection function $\Omega(t)$ which after a threshold is applied, at its peak points, gives the onset times. Activation function ReLU is used because of the lack of interest in negative changes in energy which does not contribute to onset detection.

For every frequency bin i, a function $d_i(t)$ is calculated as seen in Equation 3.11

$$d_i(t) = \sqrt{\left[ReLU\left(\frac{\theta\mu_i}{\theta t}\right)\right]^2 + \left(\frac{\theta^2\varphi_i}{\theta t}\right)^2} \tag{3.11}$$

where ReLU is defined as $f(x) = x^+ = max(0, x) = \frac{f(x)+|f(x)|}{2}$. Then the polynomial onset detection function $\Omega(t)$ is calculated by adding up all the bins along the frequency axis

$$\Omega(t) = \sum_{i=0}^{N} d_i(n) \tag{3.12}$$

where N is the number of frequency bins.

Require STFT output of narrow window
$magnitudes \Leftarrow$ convert $STFToutput$ to magnitudes
$phases \Leftarrow$ convert $STFToutput$ to phases
Form an array D with length $(STFToutput).length - 2$
$D[0] \Leftarrow 0, D[1] \Leftarrow 0$
for $i = 2$ to $(STFToutput).length$ **do**
  $diff\_m \Leftarrow magnitudes[i] - magnitudes[i-1]$
  $diff\_m \Leftarrow (|diff\_m| + diff\_m)/2$
  $diff\_p \Leftarrow phases[i] - 2 * phases[i-1] + phases[i-2]$
  $D[i] \Leftarrow \sqrt{(diff\_m)^2 + (diff\_p)^2}$
end for

Figure 3.5. Onset Detection Algorithm

3.6.1. Thresholding for Onsets

To acquire an accurate list of onset times, a peak picking algorithm (Figure 3.6) is necessary as using every peak of the $\Omega(t)$ would cause an overestimation and using too few of them would cause an underestimation of the actual number of onsets. Both cases would create a high error rate. Picking the correct peak points is possible with a dynamic threshold [4] (Equation 3.13), given that the correct constants are supplied. Each value of the dynamic threshold δ_t, for a H-length sliding analysis is given:

$$\delta_t(m) = C_t median_{\gamma_2}(w_t k_m), k_m \in \left[m - \frac{H}{2}, m + \frac{H}{2}\right] \tag{3.13}$$

where C_t is a scaling factor and w_t is a triangular windowing function of size H. w_t is used to give the values a higher weight the closer they are to the middle of the current analysis segment. This allows the shape of the thresholding function to stay similar with

the onset detection function when sudden high peaks occur. The values C_t and H adjust the thresholding for how high and steep it should be respectively.

Require onsets, half window size, threshold height constant
Form an empty array $peaks$
From an empty array $dynamic\_threshold$
$half\_h \Leftarrow$ half window size
$c1 \Leftarrow$ threshold height constant
$maximas\_indices \Leftarrow$ local maxima points from $onset$
for $i = half\_h$ to $onsets.count$ **do**
  $window \Leftarrow$ onset[$i - half\_h$ to $i + half\_h - 1$]
  $weigths \Leftarrow$ number from 1 to $half\_h$ followed by $half\_h$ to 0
  $average \Leftarrow$ weighted average of $window$ with weights $weights$
  append $(c1 * average)$ to $dynamic\_threshold$
end for
for $i = 0$ to 5 **do**
  insert 0 to $dynamic\_threshold[i]$
end for
$i \Leftarrow 0$
while $i < dynamic\_threshold.length$ and $i < onset.count$ **do**
  if $onsets[i] > dynamic\_threshold[i]$ and i is in $maximas\_indices$ **then**
    if a peak from $i - half\_h$ to $i + half\_h$ is not already selected **then**
      append i to $peaks$
    else if current peak is higher than the already selected **then**
      put new peak in place of the already selected one
    end if
  end if
  $i \Leftarrow i + 1$
end while

Figure 3.6. Peak Picking Algorithm

3.7. MIDI Conversion

Finally, knowing the notes with their octaves and durations, writing this information in a MIDI file can be done by using the semitone difference from Equation 3.10 and knowing that A_4 in MIDI is 69, a frequency value's MIDI number can be calculated with $69 + n$.

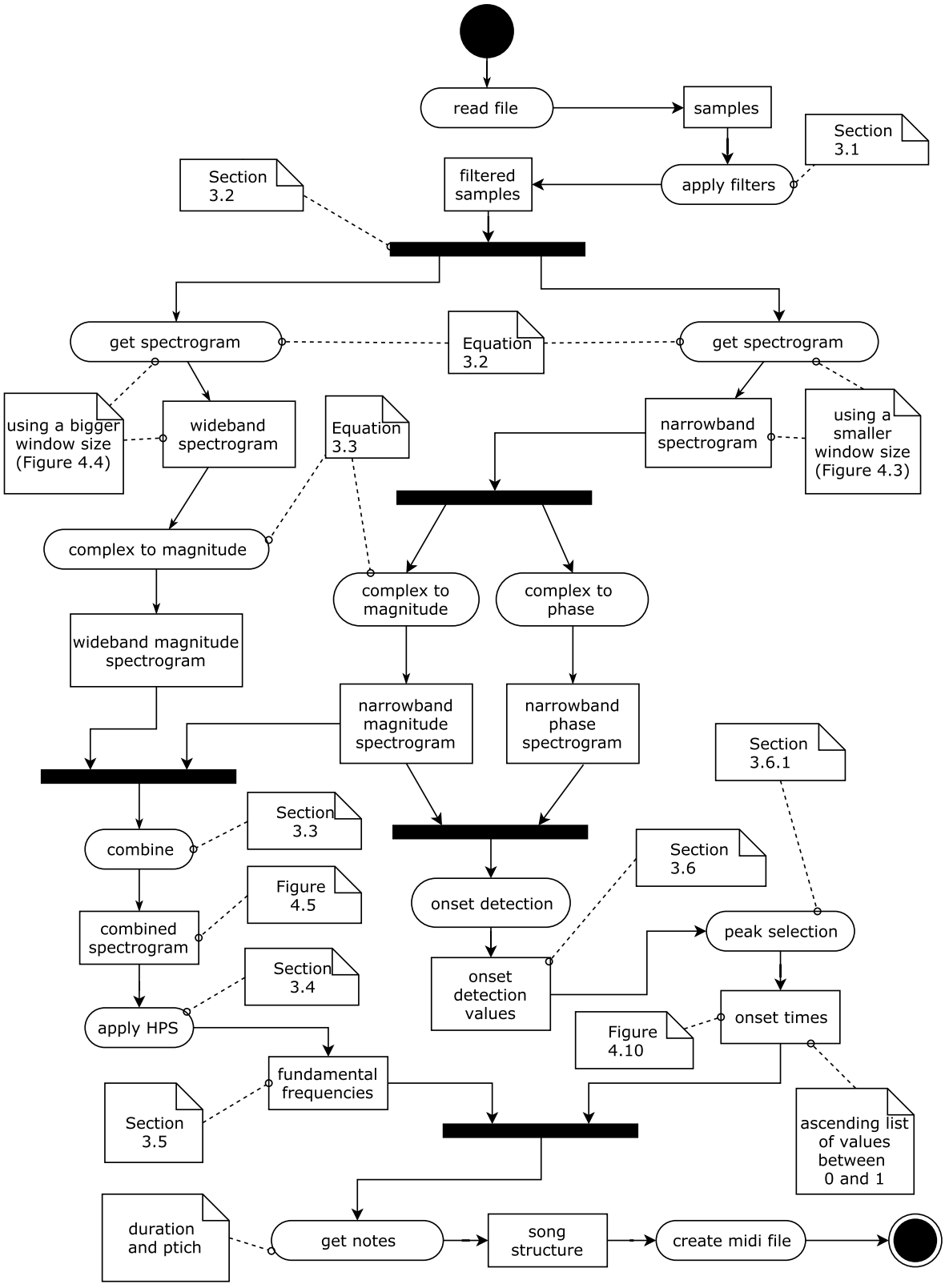

Figure 3.7. Activity Diagram

4. DESIGN AND IMPLEMENTATION

4.0.1. FFU$_{W}eb_{v}ulnerability_{s}canner.py$

4.0.1.1. Import Statements.

- The `pystyle` module is used for colorful text styling in the terminal.
- `validators` helps in validating URLs.
- `webbrowser` is used to open HTML reports in the web browser.
- Common modules such as `os`, `argparse`, `socket`, `sys`, `requests`, `datetime`, and `BeautifulSoup` are imported for various functionalities including HTTP requests, parsing URLs, handling dates and times, and parsing HTML.
- Various custom modules (`ffusubcodes` package) are imported, which seem to include functions for different vulnerability scans and checks.

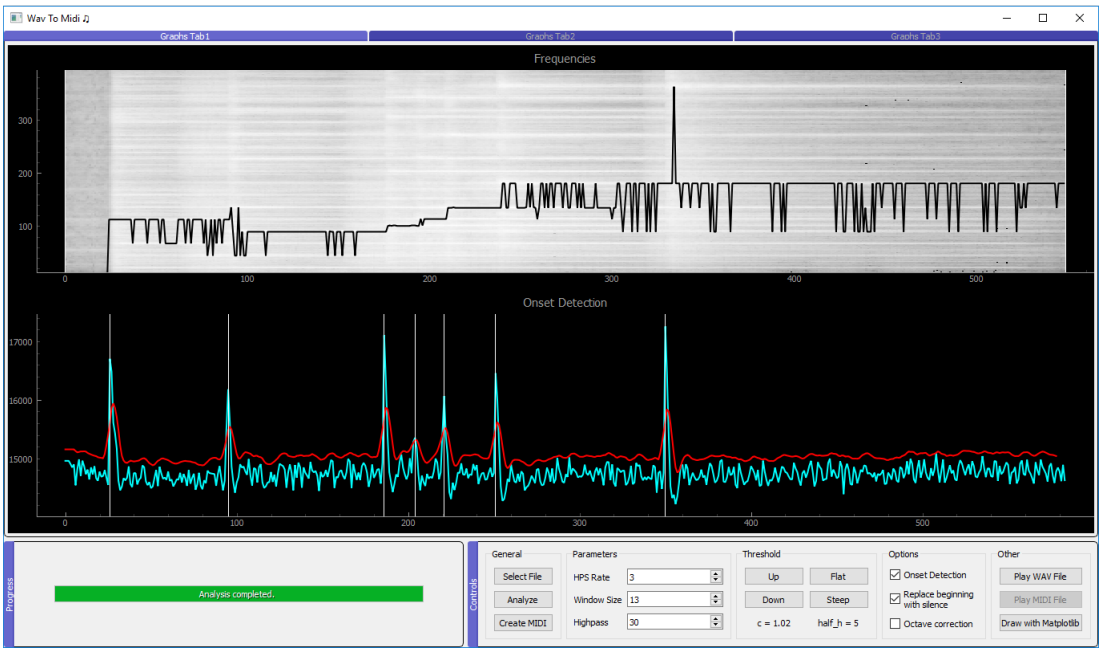

Figure 4.1. User Interface

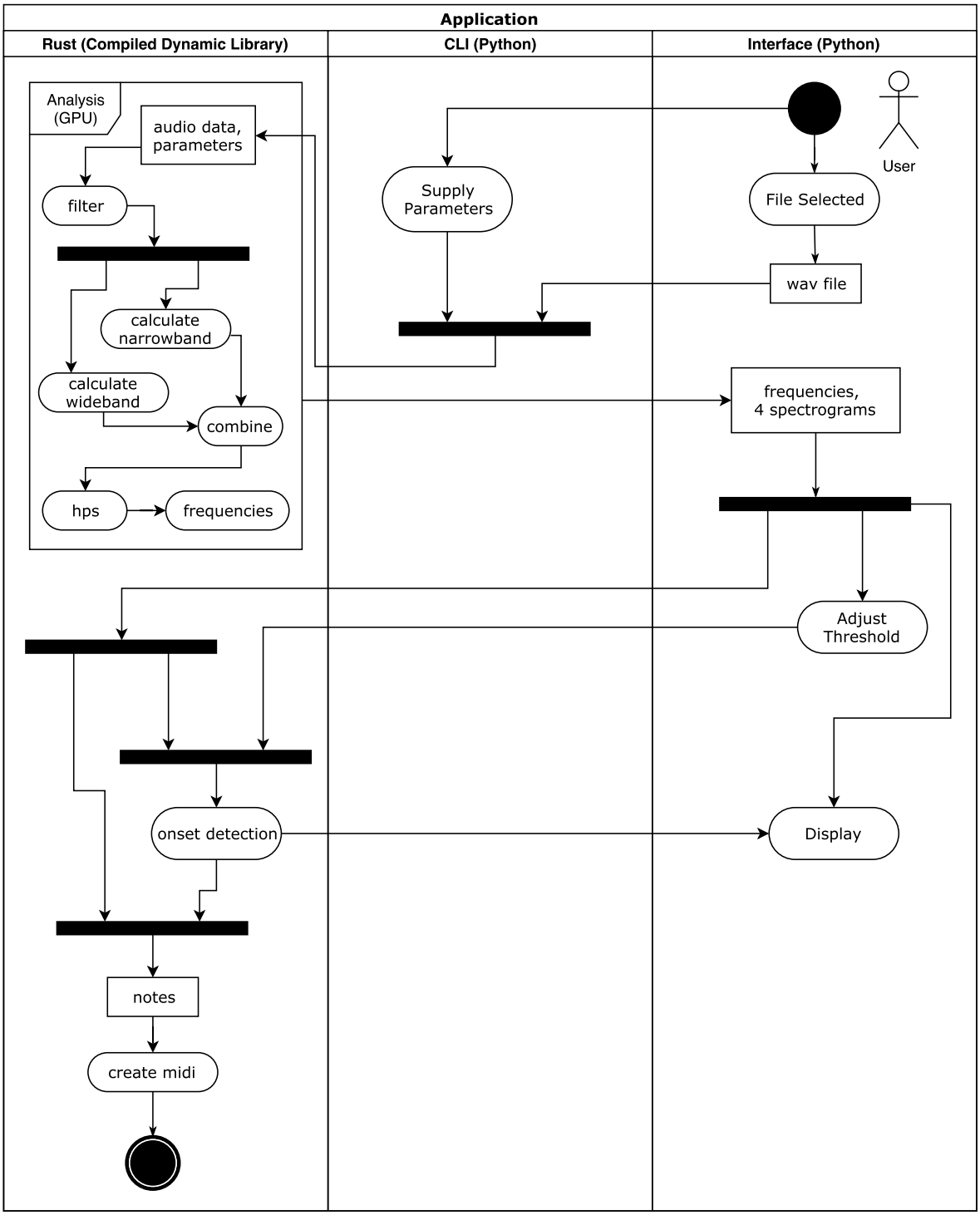

Figure 4.2. Implementation Diagram

4.0.1.2. print$_{l}ogo.$

- This function prints a stylized logo for the scanner, using red text color for emphasis.

4.0.1.3. get$_{t}arget_{u}rl.$

- This function prompts the user to enter a target URL.
- It validates the URL and, if valid, resolves the domain to an IP address.
- It creates a folder named with the domain and current date to store scan results.
- If the domain cannot be resolved, it informs the user and prompts again.

4.0.1.4. check$_{t}arget_{u}rl.$

- This function is similar to `get_target_url` but designed for checking an already provided target URL.
- It validates the URL, resolves the domain, creates a folder, and handles errors similarly.

4.0.1.5. collect$_{u}rls_{f}rom_{f}ile.$

- This function reads URLs from a specified file, returning a list of URLs.
- If the file is not found, it prints an error message.

4.0.1.6. create$_{f}older.$

- This function creates a folder with the given name if it doesn't already exist.
- It provides feedback on whether the folder was created or already exists.

4.0.1.7. get$_{l}inks_{f}rom_{p}age.$

- This function fetches a webpage and extracts all links (`href` attributes of `a` tags) on the page.
- It returns a list of fully qualified URLs.
- It handles errors in making the HTTP request.

4.0.1.8. remove$_{d}uplicate_{l}ines.$

- This function removes duplicate lines from a file, ensuring each line is unique.

4.0.1.9. crawl$_{a}ll.$

- This function recursively crawls a website starting from the `base_url`, up to a specified `max_depth`.
- It saves internal links and external links in separate files.
- It uses a nested `crawl` function to handle the recursion and keeps track of visited URLs to avoid infinite loops.
- It removes duplicate lines from the output files.

4.0.1.10. scan$_{s}ingle_{h}ost_{l}imited.$

- This function performs a limited scan on a single host.
- It runs several checks and scans (e.g., Shodan, headers, subdomains, log files, meta files).
- It crawls the website to a depth of 1, tests URLs for sensitive data, and generates an HTML report.

4.0.1.11. scan$_{s}ingle_{h}ost.$

- This function performs a comprehensive scan on a single host, including the limited scan checks and additional checks for vulnerabilities such as SQL injection, XSS, and command injection.
- It crawls the website and tests each URL for various vulnerabilities.
- It generates an HTML report of the findings.

4.0.1.12. show$_{m}enu.$

- This function prints a menu for the user to choose different scan options.

4.0.1.13. main.

- This is the main entry point of the script.
- It shows the menu, takes user input, and calls the appropriate function based on the user's choice.
- It handles scanning single targets, both limited and full scans, as well as specific vulnerability checks for given URLs.
- It loops until the user chooses to exit.

4.0.2. check$_{d}irectory_{l}isting.py$

4.0.2.1. Import Statements.

- The `requests` library is used to make HTTP requests.

4.0.2.2. main.

- The function `main` takes two arguments: `folder_name` (the name of the folder to save the output) and `url` (the target URL to scan).

4.0.2.3. Opening the Output File.

- Opens a file named directorylisting.txt in the specified folder for appending. The encoding='utf-8' ensures that the file can handle a wide range of characters.

4.0.2.4. Reading the Payloads File.

- Reads the directory listing payloads from `directorylisting_payloads.txt`. Each line in this file is a potential directory name to check on the target URL.

4.0.2.5. For.

- Iterates over each keyword (directory name) read from the payloads file.
- For each keyword, it makes a GET request to the target URL concatenated with the keyword.
- Strips any leading/trailing whitespace from the keyword to ensure a clean URL.
- Checks if the HTTP response status code is 200 OK and if the response text contains "Index of", which is a common indicator of directory listing being enabled.
- If both conditions are met, writes a message to the output file indicating that the URL is vulnerable and includes the response text.
- Catches any exceptions that occur during the HTTP request and prints the error message.
- Closes the output file after all requests have been made and results written.

4.0.3. **check**$_{d}irectory_{l}isting_{a}dminpoint.py$

4.0.3.1. Import Statements.

- The `requests` library is used to make HTTP requests.

4.0.3.2. file$_{h}as_{d}ata.$

- This function checks if a file exists and contains at least one line of data.
- It attempts to open the file and read the first line. If the file does not exist, it returns `False`.

4.0.3.3. main.

- The function `main` takes one argument: `folder_name`, which is the name of the folder containing the `admin_entry_points.txt` file.

4.0.3.4. Opening the Output File.

- Opens a file named `directorylisting_admin_page.txt` in the specified folder for appending. The `encoding='utf-8'` ensures that the file can handle a wide range of characters.

4.0.3.5. Reading the Payloads File.

- Checks if the `admin_entry_points.txt` file exists and has data. If so, it reads all lines from the file.

4.0.3.6. Making the HTTP Requests and Closing the Output File.

- Iterates through each endpoint URL and makes a GET request.
- If the response status code is `200 OK` and the response text contains "Index of", it writes a message to the output file indicating that the URL is vulnerable.
- Closes the output file after all requests have been made and results written.

4.0.4. command$_injection.py$

4.0.5. enumarate$_logfiles.py$

4.0.6. enumarate$_sensible_data.py$

4.0.7. enumerate$_forms.py$

4.0.8. find$_admin_entry_points.py$

4.0.9. header.py

4.0.10. html$_result_report.py$

4.0.11. idor.py

4.0.12. ports$_scan.py$

4.0.13. shodan.py

4.0.14. sqli.py

4.0.15. sqlicanner.py

4.0.16. subdomain.py

4.0.17. subdomain$_keyword.py$

4.0.18. web$_server_metafiles.py$

4.0.19. xssscanner.py

It is continued by converting the samples of the audio into an Array structure provided by the ArrayFire library so that the calculations can be continued on the GPU. Second thing is to filter the samples with a convolution function which was also provided within the library.

After the filtering, two spectrograms are calculated with the user supplied parameters and the filtered samples. Algorithm of the short time Fourier transform with complex valued output is as seen in Figure 3.1

Narrowband spectrogram's windows size comes from the parameter where as wideband spectrogram's is a fixed value of 44100. Wideband spectrogram is used for its characteristics of good frequency resolution and calculating it with a window size of 44100 makes every frequency bin correspond to 1 Hz. The FFT calculations take place in parallel on the GPU to utilize a highly concurrent environment. Narrowband and wideband spectrograms' characteristic properties can be seen in Figure 4.3 and Figure 4.4 respectively. Also some frequencies from harmonics are present on some time frames in both figures.

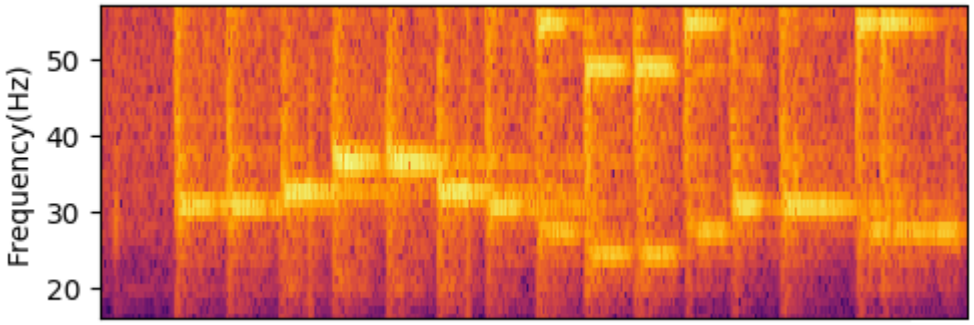

Figure 4.3. Narrowband Spectrogram

A narrowband spectrogram has good time resolution, meaning that note changing points of time are relatively more distinguishable. Therefore it is suitable for the retrieval of information about the onset times and durations of the notes.

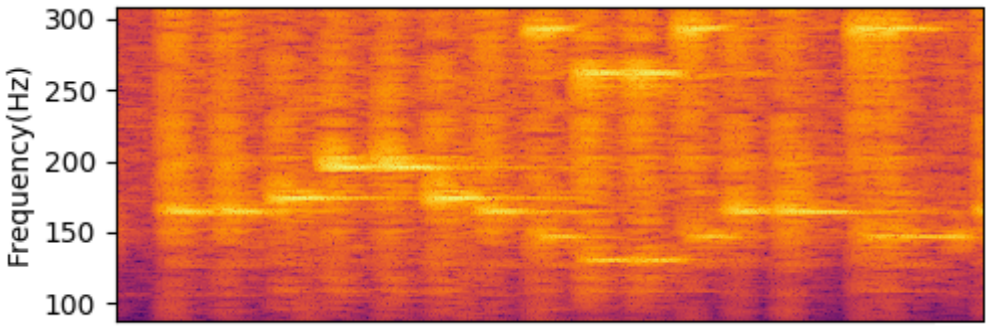

Figure 4.4. Wideband Spectrogram

A wideband spectrogram has good frequency resolution, meaning that the high amplitude frequency bins more accurately represent the actual frequencies. Therefore it is suitable for retrieval of information about the pitch.

After both spectrogram calculations are complete they are set to be combined. Combining is done by an element-wise multiplication and requires both to be the same size and shape. For resizing first and multiplying next, provided functions from the ArrayFire library were used as these matrix operations are much faster on the GPU taking advantage of parallelism. Combined spectrogram conveys the characteristic from both which can be seen in Figure 4.5

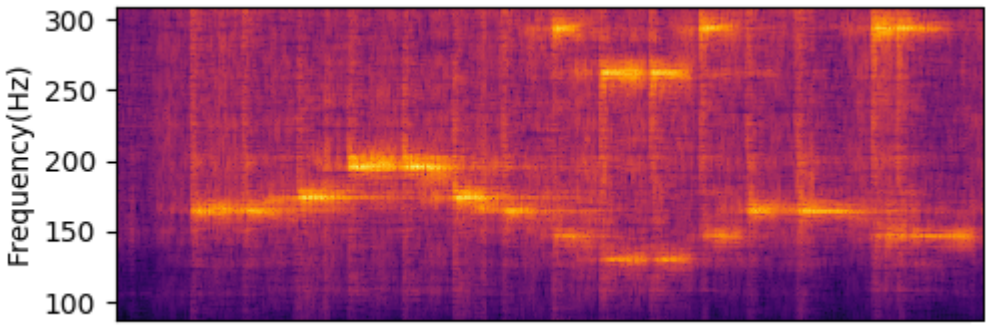

Figure 4.5. Combined Spectrogram

Then, using the Equation 3.7, HPS technique is applied to the resulting combined spectrogram. HPS algorithm can be seen in Figure 3.2

HPS applied spectrogram will have the fundamental frequencies standing out with their high amplitudes. The spectrogram can be seen in Figure 4.6 with a white line plotting the fundamentals.

Acquiring the fundamentals i.e. the frequencies with the highest amplitude for every time frame is done using the Equation 3.8 and can be implemented as seen in Figure 3.4

At this point, for some inputs, converting the fundamentals into notes by getting the pitches and durations can be done, such as inputs with no repeating notes and no fluctuations in the fundamentals list. Some violin recordings or computer generated music with

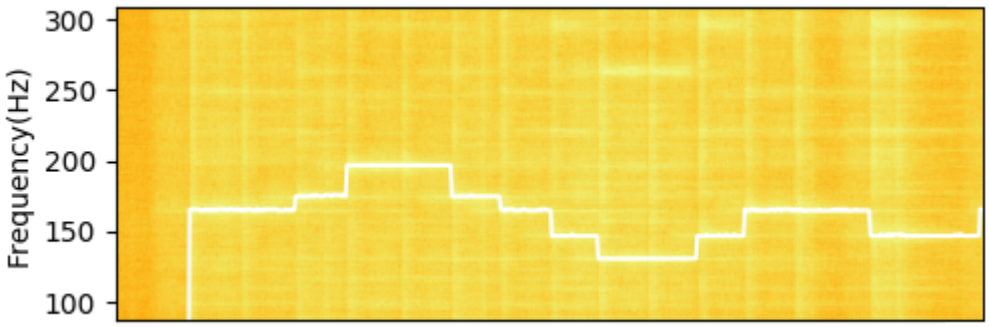

Figure 4.6. HPS Applied Spectrogram

few number of harmonics fall into this category. Otherwise the analysis should be continued with the onset detection using the Equation 3.11. On Figure 4.6 some visual changes can be observed at places where although the fundamental does not change. These are the repeated notes and will be detected with the onset detection procedure. If the process was started from the graphical user interface, it is now expected from the user to examine and adjust the threshold for onset detection if they wish to do so.

Peak points of the onset detection function which is calculated in Figure 3.5 some of which eliminated by the threshold are converted into a value between zero and one where zero is the beginning and one is the end of the audio.

Figure 4.7 shows the selected onset times where cyan is the detection function, red is the threshold and the vertical lines are the selected peaks. Between every two onset time is a duration of a note. Number of samples and the length of the audio in seconds is used to calculate a division value for the MIDI file. Division, is a unit delta-time for writing the notes to MIDI. Notes' durations are set in relation to this division value.

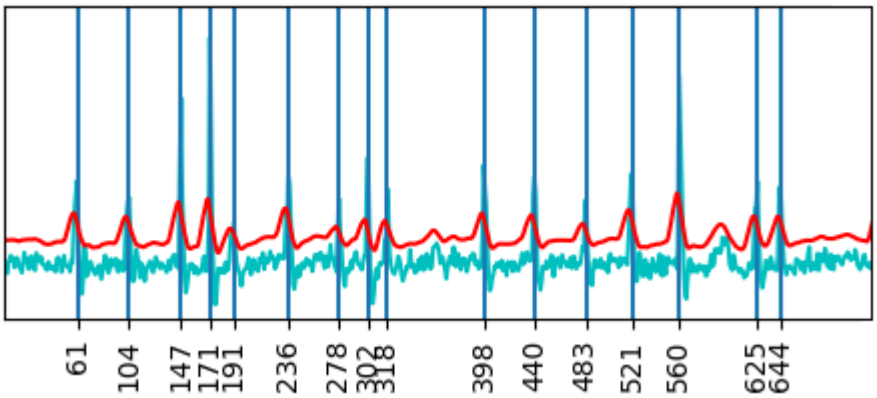

Figure 4.7. Onsets

4.1. GPU Calculations

Throughout the analysis, ArrayFire's [5] library functions are used for operations which can benefit from high concurrency. The mainly used operations are: convolution, fft, resizing matrices, interpolation and element-wise operations. The library uses an analogy where host is the CPU and the device is the GPU. Before starting the analysis, samples are put into an internal structure of the library, called *Array*. This data structure can be sent to the device and back through the system BUS and hold in buffers which are internally managed by the library. To minimize the overhead caused by the transfer between the host and the devices and remove it as a bottleneck Array type acts as a pointer, which points to a memory location on the VRAM. To reduce the overhead in the implementation Array data are sent to the device after sampling is done and not transferred back to the host to the very end until no other operations left which can benefit from high concurrency.

5. TEST AND RESULTS

Following test results are acquired from wav files that were converted from MIDI to know the actual notes and have input without noise. MIDI file is a scale run down of D minor pentatonic. Pitch accuracy is calculated by examining time slots and checking whether it matches the original. Whether the octaves are correct or not is not taken into consideration.

Table 5.1. Tests.

	Instrument	BPM	Pitch Accuracy	Additional Info
1	Guitar	100	%100	All notes 1 Octave Below Actual
2	Piano	100	%87	
3	Violin	100	%83	Onset Detection off
4	Guitar	150	%100	
5	Piano	150	%83	non-defaults c=1.03,h=5
6	Violin	150	%91	Onset Detection off
7	Guitar	100*	%100	
8	Piano	100*	%91	non-defaults HPS:4, c=1.01
9	Violin	100*	%87	Onset Detection off
10	Guitar	150*	%98	
11	Piano	150*	%86	non-defaults c=1.01
12	Violin	150*	%85	Onset Detection off
13	Guitar	200*	%75	
14	Piano	200*	%55	non-defaults c=1.01
15	Violin	200*	%65	Onset Detection off

*: denoted rows are 16^{th}, otherwise quarter notes

Instruments with low attack rates tend to give better results without onset detection. Results confirmed that the algorithm works better on recordings with definite onsets with high attacks and increasing the note rapidity affects onset detection negatively.

Results from speed tests confirmed that reimplementation of the algorithm using GPU showed to be the correct approach regarding processing time. Even though the former utilized all available CPU cores for the calculations, it was not sufficient for moderately long audio inputs and processing times were not acceptable.

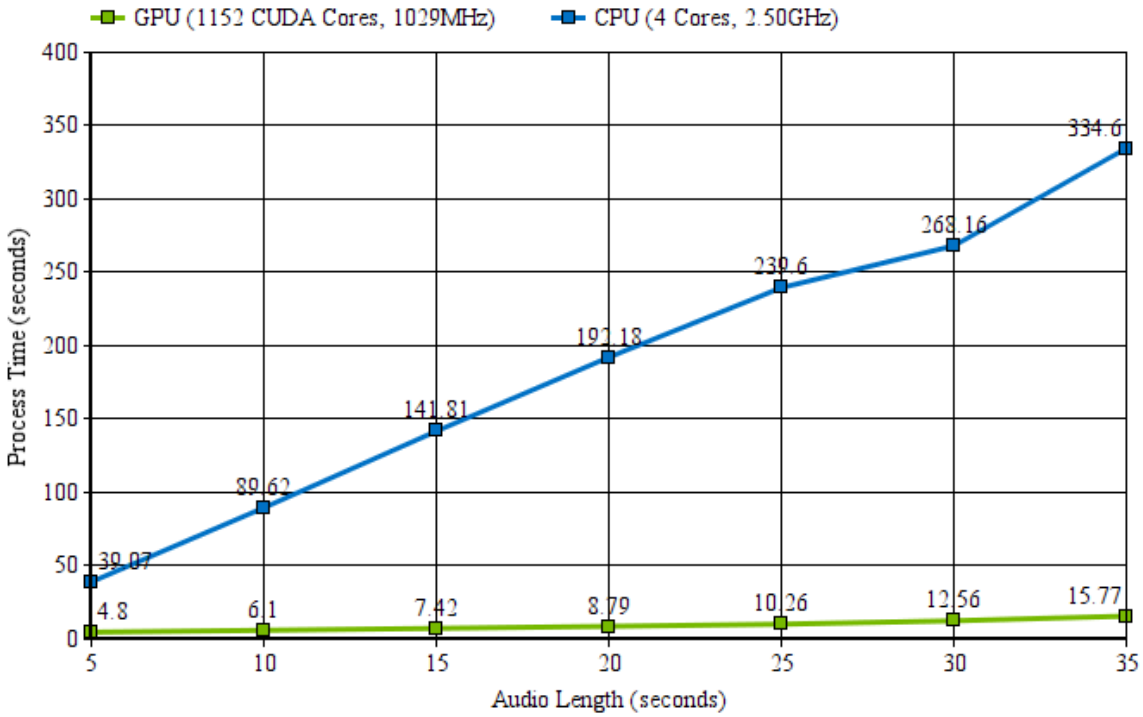

Figure 5.1. Speed Test

6. CONCLUSION

Implemented algorithm has good success rate given that the correct parameters are supplied and a monophonic audio with a high signal to noise ratio is used. Otherwise, improvements are needed.

6.1. Future Work

To improve the algorithm that currently works on monophonic inputs and extend it to work on polyphonic inputs, filter banks can be used by dividing the input into frequency channels and analysing each separately. Current limitation on input audio length infused by the memory of the video card can be solved by dividing the input into time slices and putting them in a queue before applying the same algorithm. Parameters that are currently requested from the user can be automated by implementing decision trees. Also speed tests showed that a real time implementation is possible.

Bibliography

[1] S. Cheung and J. S. Lim, "Combined multiresolution (wide-band/narrow-band) spectrogram," *IEEE Transactions on signal processing*, vol. 40, no. 4, pp. 975–977, 1992.

[2] T. Smyth. "Music 270a: Signal analysis." (2015), [Online]. Available: `http://musicweb.ucsd.edu/%7Etrsmyth/analysis/Harmonic_Product_Spectrum.html`.

[3] B. H. Suits. "Physics of music - notes." (1998), [Online]. Available: `http://pages.mtu.edu/~suits/NoteFreqCalcs.html`.

[4] C. Duxbury, J. P. Bello, M. Davies, M. Sandler, *et al.*, "Complex domain onset detection for musical signals," in *Proc. Digital Audio Effects Workshop (DAFx)*, Queen Mary University London, vol. 1, 2003, pp. 6–9.

[5] P. Yalamanchili, U. Arshad, Z. Mohammed, *et al.*, *ArrayFire - A high performance software library for parallel computing with an easy-to-use API*, Atlanta, 2015. [Online]. Available: `https://github.com/arrayfire/arrayfire`.

APPENDIX A: RUST CODE

```
//lib.rs

#![feature(step_by)]
#![allow(dead_code)]
#![allow(deprecated)]
#![feature(use_extern_macros)]

extern crate hound;
extern crate arrayfire;

use arrayfire::{Array, Dim4, Seq};

use std::f64;
use std::os::raw;
use std::ffi::CString;
use std::path::Path;
use std::slice;

mod spectrogram;
mod filters;
mod postprocess;

#[repr(C)]
pub struct FFI_Spectrogram {
    data : *const *const raw::c_double,
    shape : (u64, u64),
}

#[no_mangle]
pub fn analyze(file_name: *const raw::c_char, window_size: raw::c_uint, highpass: raw::c_uint,
               hps_rate: raw::c_uint) -> *const *const raw::c_void
{
    //println!("Active Backend: {}", arrayfire::get_active_backend());

    println!("reading file...");
    let name: String = unsafe{std::ffi::CStr::from_ptr(file_name)}.to_string_lossy().into_owned();
    let mut reader = hound::WavReader::open(name.clone()).unwrap();

    //print_audio_details(&reader);

    let data : Vec<f64> = reader.samples::<i16>().map(|sample| sample.unwrap() as f64).collect();
    let mut data_af = Array::new(&data, Dim4::new(&[data.len() as u64,1,1,1]));

    println!("applying preprocess filters..."); // TODO: filters as parameters
    data_af = ::filters::highpass(data_af, highpass as usize);
    //data_af = ::filters::lowpass(data_af, 330);

    let mut graphs = Vec::<*const raw::c_void>::new();

    println!("calculating narrowband spectrogram...");
    let window_size = (2. as f64).powi(window_size as i32) as usize;
    let complex = spectrogram::stft(&data_af, window_size, 1024);
    let narrowband = spectrogram::complex_to_magnitude(&complex);
```

```
    graphs.push(to_ffi(&spectrogram::to_host(&narrowband)));

    println!("calculating wideband spectrogram...");
    let wideband = spectrogram::get_spectrogram(&data_af, 44100, 1024);
    graphs.push(to_ffi(&spectrogram::to_host(&wideband)));

    println!("combining spectrograms...");
    let combined = spectrogram::combine(&narrowband, &wideband);
    graphs.push(to_ffi(&spectrogram::to_host(&combined)));

    println!("calculating harmonic product spectrum...");
    let hps = spectrogram::harmonic_product_spectrum(combined, hps_rate);
    graphs.push(to_ffi(&spectrogram::to_host(&hps)));

    println!("getting frequencies...");
    let frequencies = spectrogram::get_frequencies(&hps);
    let mut frequencies_host : Vec<raw::c_uint> = vec![0; frequencies.elements()];
    frequencies.host(frequencies_host.as_mut_slice());
    let freq_p = Box::into_raw(frequencies_host.into_boxed_slice()) as *const raw::c_uint;
    graphs.push(freq_p as *const raw::c_void);

    println!("onset detection function...");
    let phase = spectrogram::complex_to_phase(&complex);
    let complex_df = spectrogram::onset_detection(&narrowband, &phase);
    let detect_p = Box::into_raw(complex_df.into_boxed_slice()) as *const raw::c_double;
    graphs.push(detect_p as *const raw::c_void);

    println!("Analysis completed.");
    arrayfire::device_gc();

    Box::into_raw(graphs.into_boxed_slice()) as *const *const raw::c_void
}

fn to_ffi(s : &Vec<Vec<f64>>) -> *const raw::c_void
{
    let p_array : Vec<*const raw::c_double> =
        s.iter().map(|c| Box::into_raw(c.clone().into_boxed_slice()) as *const raw::c_double).collect();

    let spect = FFI_Spectrogram {
        shape : (s.len() as u64, s[0].len() as u64),
        data : Box::into_raw(p_array.into_boxed_slice()) as *const *const raw::c_double
    };

    Box::into_raw(Box::new(spect)) as *const raw::c_void
}

fn post_filter(spectrogram: &Array, above: usize, below: usize) -> Array
{
    let sequence0 = Seq::new(above as u32, below as u32, 1);
    let sequence1 = Seq::new(0, (spectrogram.dims()[1]-1) as u32, 1);
    let mut idxr = arrayfire::Indexer::new();
    idxr.set_index(&sequence0, 0, Some(true));
    idxr.set_index(&sequence1, 1, Some(true));

    arrayfire::index_gen(&spectrogram, idxr)
}
```

```
#[no_mangle]
pub extern fn clean2d( ptr : *const raw::c_void)
{
    unsafe
    {
        let ffi_spect = Box::from_raw(ptr as *mut FFI_Spectrogram);
        let data = std::slice::from_raw_parts(ffi_spect.data, ffi_spect.shape.0 as usize);
        for x in 0..data.len() { Box::from_raw(data[x] as *mut f64); }
    }
}

#[no_mangle]
pub extern fn clean1d( ptr : *const raw::c_void)
{
    unsafe { Box::from_raw(ptr as *mut raw::c_double) };
}

fn print_audio_details(reader: &hound::WavReader<std::io::BufReader<std::fs::File>>)
{
    print!("Audio: {{ Duration: {} seconds, ", reader.duration() / reader.spec().sample_rate);
    print!("Sample Rate: {}, ", reader.spec().sample_rate);
    print!("Channels: {}, ", reader.spec().channels);
    println!("Bit Depth: {} }}", reader.spec().bits_per_sample);
}

#[no_mangle]
pub extern fn create_midi(frequencies: *mut raw::c_uint, f_len: raw::c_uint,
                          onsets: *mut raw::c_double, o_len: raw::c_uint,
                          file_name: *const raw::c_char, onset_detection: bool) -> *const raw::c_char
{
    let name: String = unsafe { std::ffi::CStr::from_ptr(file_name).to_string_lossy().into_owned() };
    let reader = hound::WavReader::open(&name).unwrap();

    // division = 1 beat = 1 seconds
    let seconds = reader.duration() as f64 / reader.spec().sample_rate as f64;
    let division : i16 = (f_len as f64 / seconds) as i16;

    let f : &[u32] = unsafe { slice::from_raw_parts(frequencies, f_len as usize) };
    let o : &[f64] = unsafe { slice::from_raw_parts(onsets, o_len as usize) };

    let notes = postprocess::get_notes(&f.to_vec(), &o.to_vec(), onset_detection);

    let midi_name = String::from(Path::new(name.as_str()).file_stem().unwrap().to_str().unwrap());
    let song = postprocess::Song {notes: notes, name: midi_name, division: division};

    //println!("{}", song);
    let midi_filename = postprocess::write_midi(song);
    println!("writing to {}.mid", midi_filename);
    CString::new(midi_filename).unwrap().into_raw()
}
```

```
//spectrogram.rs

extern crate apodize;

use arrayfire::*;
use self::apodize::{hanning_iter};
use std::f64;

type Column = Vec<f64>;
type Spectrogram = Vec<Column>;

pub fn get_spectrogram(audio_data : &Array, window_size: usize, step_size: usize) -> Array
{
    let array = stft(audio_data, window_size, step_size);
    complex_to_magnitude(&array)
}

pub fn complex_to_magnitude(stft: &Array) -> Array
{
    let spect_len = stft.dims()[1];
    let result = Array::new_empty(stft.dims(), DType::F64);

    for index in 0..spect_len
    {
        set_col(&result, &magnitude(&col(&stft, index)), index);
    }

    return result;
}

pub fn complex_to_phase(stft: &Array) -> Array
{
    let spect_len = stft.dims()[1];
    let result = Array::new_empty(stft.dims(), DType::F64);

    for index in 0..spect_len
    {
        set_col(&result, &phase(&col(&stft, index)), index);
    }

    return result;
}

pub fn stft(audio_data : &Array, window_size: usize, step_size: usize) -> Array
{
    let audio_len : usize = audio_data.elements();

    let spect_len : u64 = (( audio_len-(window_size-step_size) ) / step_size) as u64;
    let array = Array::new_empty(Dim4::new(&[window_size as u64, spect_len, 1,1]), DType::C64);

    let hanning_len = Dim4::new(&[window_size as u64,1,1,1]);
    let hanning = Array::new(&hanning_iter(window_size as usize).collect::<Vec<f64>>(), hanning_len);

    for (index, start) in (0..audio_len-window_size).step_by(step_size).enumerate()
    {
        let sequence =  Seq::new(start as u32, start as u32 + (window_size-1) as u32, 1);
        let mut idxr = Indexer::new();
```

```
        idxr.set_index(&sequence, 0, Some(true));

        let new_col = fft(&(index_gen(&audio_data, idxr) * hanning.clone()), 1., window_size as i64);
        set_col(&array, &new_col, index as u64);
    }

    return get_half(&array);
}

fn get_half(stft: &Array) -> Array
{
    let sequence0 = Seq::new(0, ((stft.dims()[0] as f32 / 2.).floor()-1.) as u32, 1);
    let sequence1 = Seq::new(0, (stft.dims()[1]-1) as u32, 1);
    let mut idxr = Indexer::new();
    idxr.set_index(&sequence0, 0, Some(true));
    idxr.set_index(&sequence1, 1, Some(true));

    index_gen(&stft, idxr)
}

fn magnitude(col : &Array) -> Array
{
    let magnitude = sqrt( &(pow(&real(&col),&2, true) * pow(&imag(&col),&2, true)) );
    floor(&log1p(&magnitude))
}

fn phase(col: &Array) -> Array
{
    atan2(&imag(col), &real(col), true)
}

pub fn onset_detection(m: &Array, p: &Array) -> Vec<f64>
{
    let mut result : Vec<f64> = Vec::new();

    let spect_len = m.dims()[1];
    for index in 2..spect_len
    {
        let predicted_m = col(&m, index-1);
        let predicted_p = mul(&col(&p, index-1),&2, true) - col(&p, index-2);

        let a = col(&m, index) - predicted_m;
        let rectified = (&a + abs(&a))/2;
        let b = col(&p, index) - predicted_p;
        let distance = sqrt(&(pow(&rectified, &2, true) + pow(&b, &2, true)));

        result.push(sum_all(&distance).0);
    }

    let mut v = vec![result[0];2];
    v.extend(result);
    return v;
}

pub fn combine(narrowband : &Array, wideband : &Array) -> Array
{
    let wd = wideband.dims();
```

```
    let narrow = resize(&narrowband, wd[0] as i64, wd[1] as i64, InterpType::BILINEAR);
    mul(wideband, &narrow, true)
}

pub fn harmonic_product_spectrum(combined : Array, rate : u32) -> Array
{
    let sequence0 = Seq::new(0 as u32, ((combined.dims()[0] as f32 / rate as f32)-1.) as u32, 1);
    let sequence1 = Seq::new(0 as u32, (combined.dims()[1]-1) as u32, 1);
    let mut idxr = Indexer::new();
    idxr.set_index(&sequence0, 0, Some(true));
    idxr.set_index(&sequence1, 1, Some(true));

    let mut hps = index_gen(&combined, idxr);

    for r in 2..rate+1
    {
        let scale = (combined.dims()[0] as f32 / r as f32) as i64;
        let downsampled = resize(&combined, scale, combined.dims()[1] as i64, InterpType::BILINEAR);

        let mut idxr = Indexer::new();
        idxr.set_index(&sequence0, 0, Some(true));
        idxr.set_index(&sequence1, 1, Some(true));
        hps *= index_gen(&downsampled, idxr);
    }

    return log1p(&hps);
}

pub fn get_frequencies(spectrogram: &Array) -> Array
{
    imax(&spectrogram, 0).1
}

pub fn to_host(spectrogram_af : &Array) -> Spectrogram
{
    let window_size = spectrogram_af.dims()[0];
    let slice_len = window_size * spectrogram_af.dims()[1];
    let mut host = vec![0.; slice_len as usize];
    spectrogram_af.host::<f64>(host.as_mut_slice());

    let mut spectrogram : Spectrogram = Vec::new();
    for column in host.chunks(window_size as usize)
    {
        spectrogram.push(column.to_vec());
    }

    return spectrogram;
}
```

```
//postprocess.rs

extern crate rimd;

use std::{f64, fmt};
use self::rimd::{ TrackEvent, Event, MidiMessage, SMFWriter, SMFFormat, Track, SMF};
use std::path::Path;
use std::collections::HashMap;

pub struct Note {name: String, midi: u8, duration: u64}
pub struct Song{pub notes: Vec<Note>, pub division: i16, pub name: String}

impl fmt::Display for Song {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        let all_notes : String =
            self.notes.iter().fold(String::new(), |s, n|
                format!("{}{}", s, format!("{}[{}] ", n.name, n.duration)));
        write!(f, "{}: {}", self.name, all_notes)
    }
}

fn get_pitch(frequency : u32) -> (String, u8)
{
    if frequency < 16 { return (String::from("R"), 0); }

    let notes = vec![ "C", "C#/Db", "D", "D#/Eb", "E", "F", "F#/Gb", "G", "G#/Ab", "A", "A#/Bb", "B" ];

    //distance to A440 of the note
    let log_base : f64 = (2.0 as f64).powf(1.0/12.0);
    let mut offset : i32 = ((frequency as f64 / 440.0).log(log_base)).round() as i32;

    let midi = 69 + offset; // A4 in midi is 69

    //adjust offset for the array
    offset += if offset >= 0 {9} else {-2};
    let note_index : usize = if offset >= 0 {offset%12} else {11 + offset%12} as usize;
    return (format!( "{note}{octave}", note = notes[note_index], octave = 4 + offset/12), midi as u8);
}

pub fn get_notes(frequencies: &[u32], onsets: &[f64], no_onset: bool) -> Vec<Note>
{
    let vec : Vec<(String, u8)> = frequencies.iter().map(|freq| get_pitch(*freq)).collect();
    let mut notes : Vec<Note> = Vec::new();

    if no_onset
    {
        let mut last: usize = 0;
        for i in 1..vec.len()
        {
            if vec[i] != vec[i-1] || i == vec.len()-1 {
                notes.push( Note {
                    name: vec[i-1].0.clone(), midi: vec[i-1].1.clone(),
                    duration: (i-last) as u64
                } );
                last = i;
            }
        }
```

```
    }
    else
    {
        let mut onsets_clone : Vec<f64> = onsets.clone().to_vec();
        onsets_clone.push(1.);
        let mut previous_onset : f64 = 0.;
        for &onset in &onsets_clone
        {
            let start_index : usize = (frequencies.len() as f64 * previous_onset).round() as usize;
            let end_index : usize = (frequencies.len() as f64 * onset).round() as usize;

            let interval : &[(String,u8)] = &vec[start_index..end_index];

            let mode = mode(interval);
            let duration : u64 = end_index as u64 - start_index as u64;

            notes.push( Note {
                name: mode.0.clone(), midi: mode.1.clone(),  duration: duration
            });

            previous_onset = onset;
        }
    }


    return notes;
}

fn mode(elements: &[(String,u8)]) -> &(String,u8) {

    let mut occurrences = HashMap::new();

    for value in elements {
        *occurrences.entry(value).or_insert(0) += 1;
    }

    occurrences.into_iter()
        .max_by_key(|&(_, count)| count)
        .map(|(val, _)| val)
        .expect("Cannot compute the mode of zero elements")
}

pub fn write_midi(song: Song) -> String
{
    let mut midi_filename = song.name.clone();
    let mut events : Vec<TrackEvent> = Vec::new();

    let mut previous_duration = 0;
    for note in song.notes.iter()
    {
        if note.name != "R"
        {
            events.push( TrackEvent {
                event: Event::Midi(MidiMessage::note_on(note.midi, 125, 0)),
                vtime: previous_duration
            } );
```

```
            events.push( TrackEvent {
                event: Event::Midi(MidiMessage::note_off(note.midi, 125, 0)),
                vtime: note.duration
            } );
        }

        if note.name == "R" {
            previous_duration += note.duration;
        } else {
            previous_duration = note.duration as u64;
        }

    }

    let tracks = vec![ Track {copyright: None, name: None, events: events} ];
    let smf = SMF {format: SMFFormat::Single, tracks: tracks, division: song.division};
    let writer = SMFWriter::from_smf(smf);

    let result = writer.write_to_file(Path::new(format!("{}{}", song.name,".mid").as_str()));
    if result.is_err()
    {
        let song_ = Song {notes: song.notes, name: song.name+"_", division: song.division};
        midi_filename = write_midi(song_);
    }

    return midi_filename;
}

//filters.rs

extern crate synthrs;
use self::synthrs::filter::{cutoff_from_frequency, highpass_filter, lowpass_filter};
use arrayfire::{Array, Dim4, convolve1, ConvMode, ConvDomain};

pub fn highpass(signal: Array, h: usize) -> Array
{
    let highpass = highpass_filter(cutoff_from_frequency(h as f64, 44100), 0.01);
    let filter : Array = Array::new(&highpass, Dim4::new(&[highpass.len() as u64,1,1,1]));
    convolve1(&signal, &filter, ConvMode::DEFAULT, ConvDomain::FREQUENCY)
}

pub fn lowpass(signal: Array, l: usize) -> Array
{
    let lowpass = lowpass_filter(cutoff_from_frequency(l as f64, 44100), 0.01);
    let filter : Array = Array::new(&lowpass, Dim4::new(&[lowpass.len() as u64,1,1,1]));
    convolve1(&signal, &filter, ConvMode::DEFAULT, ConvDomain::FREQUENCY)
}
```

```
# Cargo.toml

[package]
name = "rust_test"
version = "0.1.0"
authors = ["utku <utkuce@gmail.com>"]

[lib]
name = "mylib"
crate-type = ["dylib"]

[dependencies]
hound = "3.1.0" # read wav file
rimd = "0.0.1" # midi
arrayfire = "3.5.0" # gpu acceleration
apodize = "*" # windowing function

[dependencies.synthrs]
git = "https://github.com/gyng/synthrs" # filters
```

APPENDIX B: PYTHON CODE

```
#analyze.py

from ctypes import cdll, c_void_p, c_double, c_uint, cast
import argparse
import time
import internal_utility as iu

parser = argparse.ArgumentParser(description='Converts a monophonic wav file into MIDI')

parser.add_argument('-f', '--file-name',
    help='The file that is to be analyzed',
    required=True, type=str)

parser.add_argument('-w','--window',
    help='2 raised to the w will be used as window length for narrowband spectrogram',
    required=False, default=13, type=int)

parser.add_argument('-p', '--highpass',
    help='highpass filter frequency for preprocessing',
    required=False, default=30, type=int)

parser.add_argument('-r', '--hps-rate',
    help='the rate used for harmonic product spectrum',
    required=False, default=3, type=int)

parser.add_argument('-o', '--onset-window',
    help='half the length of the sliding window used for onset detection thresholding',
    required=False, default=5, type=int)

parser.add_argument('-c', '--threshold-constant',
    help='constant value for scaling the threshold for onset detection',
    required=False, default=1.05, type=float)

parser.add_argument('-i', '--from-interface',
    help='flag to determine if the script is called from the interface',
    action='store_true')

parser.add_argument('-n', '--no-onsets',
    help='flag to use onset detection or not', action='store_true')

args = vars(parser.parse_args())

start_time = time.time()

mylib = cdll.LoadLibrary('target\debug\mylib.dll')
mylib.analyze.restype = c_void_p

results = mylib.analyze(args['file_name'].encode('UTF-8'), args['window'],
                        args['highpass'], args['hps_rate'])

graphs = iu.from_ffi(results)
```

```
if not args['from_interface']:

    (frequencies, detection) = (graphs[4],  graphs[5])

    (peaks, threshold) = iu.peaks(detection, args['onset_window'], args['threshold_constant'])
    onsets = [ i / len(detection) for i in peaks ]

    print("Finished in %s seconds" % int(time.time() - start_time))

    mylib.create_midi((c_uint * len(frequencies))(*frequencies), len(frequencies),
                    (c_double * len(onsets))(*onsets), len(onsets),
                    args['file_name'].encode('UTF-8'), args['no_onsets'])

    print("Finished in %s seconds" % (time.time() - start_time))
    exec(open('spectrogram.py').read(), globals(), locals())

else:

    import pickle
    with open('results.temp', 'wb') as f:
        pickle.dump( graphs, f )

mylib.clean2d.argtypes = [c_void_p]
mylib.clean1d.argtypes = [c_void_p]

for i,g in enumerate(iu.pointerList.pointers):
    ptr = cast(g, c_void_p)
    mylib.clean2d(ptr) if i < 4 else mylib.clean1d(ptr)



# internal_utility.py

from __future__ import division
import numpy as np
from ctypes import *

class C_Tuple(Structure):
    _fields_ = [("x", c_uint64), ("y", c_uint64)]

class FFI_Spectrogram(Structure):
    _fields_ = [("data", POINTER(POINTER(c_double))), ("shape", C_Tuple)]

class PointerList(Structure):
    _fields_ = [("pointers", c_void_p*6)]

pointerList = PointerList()

def from_ffi(results):

    global pointerList
    pointerList = cast(results, POINTER(PointerList)).contents

    spect_list = []

    for i,g in enumerate(pointerList.pointers):
        if i < 4:
            spect_list.append(get_result(g))
```

```
        elif i == 4:
            pointer = cast(g, POINTER(c_uint*len(spect_list[3])))
            spect_list.append(np.fromiter(pointer.contents, dtype=np.uint))
        else:
            pointer = cast(g, POINTER(c_double*(len(spect_list[0]-1)) ))
            spect_list.append(np.fromiter(pointer.contents, dtype=float))

    return spect_list

def get_result(spect_pointer):

    result = cast(spect_pointer, POINTER(FFI_Spectrogram)).contents

    data = []

    for x in range(0, result.shape.x):
        array_pointer = cast(result.data[x], POINTER(c_double*result.shape.y))
        a = np.frombuffer(array_pointer.contents, dtype=float)
        data.append(a)

    return np.asarray(data)

def maximas(data):

    derivative = np.diff(data)
    maximas = []

    for i in range(1, len(data)-1):
        if data[i] - data[i-1] > 0 and data[i] - data[i+1] > 0:
            maximas.append(i)

    return maximas

def peaks(onset, half_h, c1):

    peaks = []
    maximas_indices = maximas(onset)
    dynamic_threshold = []

    for i in range(half_h, len(onset)-half_h):
        window = onset[i-half_h: i+half_h-1]
        weights = list(range(1,half_h,1)) + list(range(half_h,0,-1))
        dynamic_threshold.append(c1*np.average(window, weights=weights))

    for i in range(5):
        dynamic_threshold.insert(0, dynamic_threshold[0])

    i = 0
    while i < len(dynamic_threshold) and i < len(onset):
        if onset[i] > dynamic_threshold[i] and i in maximas_indices:
            if set(range(i-half_h, i+half_h)).isdisjoint(peaks):
                peaks.append(i)
            else:
                for (j,p) in enumerate(peaks):
                    if p in list(range(i-half_h, i+half_h)) and onset[i]>onset[p]:
                        peaks[j] = i
        i+=1
```

```
    return (peaks, dynamic_threshold)

def repeated_notes(peaks, changes, half_h, index_scale):

    repeats = []
    for p in peaks:
        l = range(round((p-half_h)*index_scale), round((p+half_h)*index_scale),1)
        if set(l).isdisjoint(changes):
            repeats.append(p)

    return repeats

def octave_correction(frequencies, wideband):

    corrected = []
    for (index,f) in enumerate(frequencies):

        if f == 0:
            corrected.append(0)
            continue

        i = 2
        current_max = f
        while f*i < wideband.shape[1]/2:
            if wideband[index][f*i] > wideband[index][current_max]:
                current_max = f*i
            i += 1

        corrected.append(current_max)

    return corrected
```